Efficient Implementation of XPath Processor on Multi-Core CPUs

Martin Kruliš and Jakub Yaghob

Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
{krulis, yaghob}@ksi.mff.cuni.cz

Abstract. Current XPath processors use direct approach to query evaluation which is quite inefficient in some cases and usually implemented serially. This may be a problem in case of processing complex queries on large documents. We propose algorithms and XML indexing techniques which are more efficient and which can utilize standard parallel templates. Our implementation is highly scalable and outperforms common XML libraries.

Key words: XML, XPath, parallel, multi-threaded, multi-core

1 Introduction

The course of processor developement has changed significantly in the past few years. Pursuit of core frequency is not as important as before. Multiple processor cores are integrated on a single chip, making parallel architectures available to common users. This trend requires programmers to focus on problem parallelization instead of classic linear optimization. Common tools for querying XML documents do not exploit computational power of multi-core CPUs. Even though this is not a problem in case of small documents and simple queries, processing complex queries on large amount of data may benefit from parallelization.

In this paper, we will focus solely on processing single query on a single document, which we expect to be loaded into main memory in DOM [7] and indexed. This restriction seems very strong, however, mainstream computers of the day can accomodate XML documents with hundreds of millions of elements in RAM. Furthermore, we assume the following about the processed XML documents:

- Documents are shallow [18]. Element nesting depth does not exceed $\mathcal{O}(\log N)$, where N denotes number of elements.
- Strings (element names, contents, etc.) are short. We expect that all strings have $\mathcal{O}(1)$ length. This is quite strong assumption, however, we are focusing on processing the element structure, not on string algorithms.
- Documents contain only small number of element and attribute identifiers respectively to total number of elements and attributes.

Most XML documents have the properties described above, so our implementation will process them just fine. Documents that do not conform to our assumptions will not be considered in this paper for the sake of scope.

XPath specification [6] contains a lot of details, many of which are uninteresting from our perspective. We will focus on processing location paths, since they present the most challenging part of XPath. Common implementations [10][11] take direct approach which can lead to inefficient algorithms. We demonstrate the problem on an example. Consider simple location path:

`/descendant::a/following::b[`P`]`

Let us assume that all elements `b` are in ‘following’ relation with any element `a`. The `following::b` step then yields the same set of nodes for every element `a` produced by previous step. Therefore, predicate P will be evaluated for each node `b` $\mathcal{O}(N_a)$ times, where N_a denotes total number of elements `a`.

Straightforward implementation leads to an algorithm which has its time complexity exponentially dependent on the level of predicate nesting. Fortunately, this problem can be avoided by introducing vector operations and min-context rules [2]. When each predicate is processed for whole vector of contexts instead of traditional recursive evaluation, the exponential phantom menace is disposed of and evaluation time remains polynomial in both size of the document and query nesting depth.

The paper is organized as follows. Section 2 revises related work. Section 3 describes XML representation and indexing techniques. Section 4 inspects proposed algorithms in detail. Section 5 suggests possible parallelism exploitation. Section 6 presents experimental results and Section 7 concludes.

2 Related Work

Efficient implementation of XPath has been addressed in several papers. One of the most significant works was presented by Gottlob et al. [2]. It proposes algorithms for processing XPath queries in polynomial time. XPath also relies on efficient evaluation of location axes which was addressed in [16]. The topic of XML processing parallelization is relatively new. First studies focused on XML parsing [14][15] and processing in a general way. Successful approach is to employ work stealing [3] in order to reduce load balancing problems.

To our best knowledge, the only work directly related to parallel processing of XPath queries was presented by IBM [4]. The paper proposes data, query and hybrid partition strategies, how to divide workload among multiple threads. The partitioned queries are then processed by Xalan. This technique can be improved [5] by introducing metrics for processing costs. These metrics determine the best points for partitioning.

We take different approach. Our strategy is to design algorithms which will utilize parallel templates and ideas presented by Intel TBB [12]. These templates implement solutions for common patterns in parallelism such as parallel-for or parallel-scan algorithms.

3 XML Index

XPath processor expects XML document to be loaded in main memory in abstract tree structure (for example DOM). This structure provides basic operations for traversing the document such as finding children, parent, or attributes of given node. Even though these operations are sufficient to traverse the whole tree and process XPath query, they are quite ineffective for complex operations.

When processing XPath query, we are often required to retrieve all nodes with specific name or of specific type that are in relation (defined by used axis) with initial node-set. This task can be accelerated significantly when proper index is built. We have used the following two indices in our implementation: the Left-right-depth index and the element index.

The *Left-right-depth* index (also denoted LRD) is often used by many XML algorithms and data structures [2][16][17] to determine the relation of two nodes in constant time. The index is based on tagging nodes of DOM structure with three integers - *left*, *right*, and *depth* value. We have tagged only element nodes and attributes as other nodes are irrelevant for XPath evaluation. We denote $l(v)$, $r(v)$, and $d(v)$ the left-value, right-value, and depth of the node v respectively.

The *element index* is formed by hash table where element names are keys. Each name holds a list of references to the DOM structure pointing to all elements with this name as depicted on Figure 1.

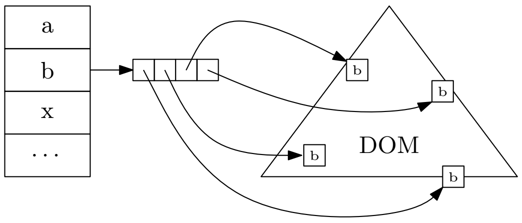

Fig. 1. Element index lists all elements of given name.

Links in the reference list are ordered by the position (i.e. the left-value) of the nodes to which they point to. This ordering is very useful, as shown later.

3.1 XPath Axes Implementation

Let us have a look on each axis and how it can benefit from these indices. In following descriptions, we expect to have context node c and location step `axis::x` where the `axis` will vary. In every case, we extract the reference list from element index using `x` as key, so we have list of all `x` elements at our disposal.

Implementation of the *descendant* axis is quite simple. The index reference list is ordered by left-values of the nodes. Furthermore, we know that every descendant of context element c has its left-value in range $(l(c), r(c)\rangle$. Such nodes are definitely stored in continuous subrange of the reference list. A binary-search algorithm can be applied to find both beginning and end of the subrange.

The *following* axis uses the same approach. If we have an element c, all following nodes have their left-values greater than $r(c)$. We simply find the first one and then take all nodes from its position to the end of the list.

The *preceding* axis is a little more complicated than the previous two. In order to determine whether node u precedes context c, we need to compare $r(u)$ with $l(c)$. Unfortunately, the reference list is not ordered by right-values. As the right-value is always greater or equal left-value (of the same node), we will use all nodes with left-value lesser than $l(c)$ as *candidates*. These candidates contain two types of nodes – predecessors and ancestors of c. We can test right-value of each candidate and filter out the predecessors.

We might create second element index where references will be ordered by right-values and implement evaluation of the *preceding* axis analogically to the *following* axis. However, we did not use it in our implementation.

The *ancestor* axis cannot be effectively accelerated by this index. On the other hand, we expect that XML documents are shallow, therefore the best way to find all ancestors of context node is to follow parental links in XML tree structure and filter nodes one by one.

Techniques described for *descendant*, *following*, and *preceding* axes could be extended also to *child*, *following-sibling*, and *preceding-sibling* relations. In order to do that, an element index must be also built for every node with children. Such index contains only children elements of associated node. The implementation would be analogical.

4 Algorithms

Before we describe optimization and implementation details of our XPath processor, we will revise recursive algorithm for location path evaluation. A location path consists of sequence of location steps. Each step takes node-set produced by previous step (called *initial node-set*) and generates another node-set which is used by successive step. First step uses the context node as a singleton and last step produces node-set which is also the result of the whole location path.

The recursive algorithm evaluates location steps as follows. First, an intermediate node-set S_i is generated for every node v_i in the initial set by application of location axis on v_i and node test to filter out nodes of specific type or name[1]. Each set S_i is filtered by predicates and S'_i ($S'_i \subseteq S_i$) is produced. Finally, all S'_i sets are united and the union is yielded as the result of the location step.

When a node-set is filtered by a predicate, the predicate is executed recursively for every node in the set using the node, its position and size of the set as context values. Result of the predicate is converted into Boolean value, and if *true*, the tested node is included in the result. A location step may have more than one predicate. In that case, predicates are applied one by one, each producing another node-set which is handed over to successive predicate.

[1] In our case, we focus only on name filters.

As mentioned before, the recursive algorithm is not very efficient. Now, we describe the most important optimizations used in our implementation.

4.1 Minimal Context Optimizations

First, we define *context flags* for each XPath expression (and subexpression):

- flag n for context node,
- flag p for position,
- and flag s for size.

These flags indicate, which parts of evaluation context are required by the expression. Expression E is tagged with set of flags denoted $cf(E) \subseteq \{n, p, s\}$. When a flag is present in the set $cf(E)$, corresponding part of context is required for evaluation of E. Formally, we define *cf* as shown in Table 1.

expression E	context flags $cf(E)$
arithmetic operators $(+, -, *, \text{div}, \text{mod})$	$cf(operand_1) \cup cf(operand_2)$
comparisons $(<, \leq, >, \geq, =, \neq)$	$cf(operand_1) \cup cf(operand_2)$
logical operators (and, or)	$cf(operand_1) \cup cf(operand_2)$
absolute location path	$\{\}$
relative location path	$\{n\}$
location step	$\{n\}$
$[predicate]$	$cf(predicate)$
union $(\vert)$	$cf(operand_1) \cup cf(operand_2)$
$position()$	$\{p\}$
$last()$	$\{s\}$
literals and functions without arguments	$\{\}$
functions with arguments	$\bigcup cf(a), a \in arguments$

Table 1. Formal definition of context flags

Even though there are eight (2^3) types of expressions depending on which of n, p, s flags they have, we will streamline the situation by recognizing only three *context classes*:

- *Context-free* class represents expressions with empty *cf* set. These expressions are either constant (like literals) or contain absolute location paths.
- Expressions in *context-node* class requires only context node, thus their $cf(E) = \{n\}$. Typical representants are relative location paths and location steps.
- Finally, we place all remaining expressions in *full-context* class.

We call the context-free class the *weakest*, because empty context flag set is always a subset of any other class. Analogically, the full-context class will be the *strongest*. Expression with non-empty $cf \subseteq \{p, s\}$ is also considered to be *stronger* than context-node expression, as the context position and size is always generated together with nodes, hence node flag is in fact implicitly included.

When evaluating location step, many nodes can be filtered by predicates multiple times if they appear in more than one intermediate (S_i) set. This may be unavoidable since the node can be at different positions or the S_i sets can have different sizes, thus they create a different context for the predicate evaluation. However, if there are no predicates which require position or size context value, the filtering procedure can be optimized.

Formally, if all predicates of a location step are from context-node or context-free class, we may unite all S_i sets before they are filtered by predicates. Then the union is filtered instead, thus predicates are executed for each node at most once. Furthermore, if there are some predicates from context-free class, we need not evaluate them for every node, as it is sufficient to execute them just once. If they resolve *true*, they can be removed from predicate list since they do not restrict the result set in any way. Otherwise (if they turn *false*), we need not resolve the location step nor the remaining part of the location path any more as it will never yield any nodes.

4.2 Vectorization of Axes

Previous optimization can be used to improve processing of axes in location steps. When axis is processed, single node is taken as input and a set of all conforming nodes is returned. If there are no full-context predicates in the location step, the intermediate node-sets (S_i) are united right after the axis part of the step (with the name filter) is resolved. Knowing this, we can optimize axis processing so it will not retrieve intermediate set for each node, but rather use entire initial node-set as input and yield the union of node-sets S_i. In following algorithms we expect that a node-set is represented by an array of nodes where nodes are ordered by their left-values (i.e. in the document order).

Descendant axis utilizes following observations: descendants of a single node are stored in continuous subrange of element index (as described in Section 3) and descendant sets of two following nodes have empty intersection. Furthermore, if we have initial nodes u and v where v is descendant of u, descendants of v are also descendants of u, therefore we need to process only node u when retrieving descendants. The algorithm will work as follows.

```
**for** $i = 1$ to $|S|$ **do**
  **if** $i = 1$ *or not*($S[i]$ descendant of *last*) **then**
    *last* $\leftarrow S[i]$
    append descendants of $S[i]$ to the result
  **end if**
**end for**
```

Following and preceding axes can be accelerated greatly by this optimization. First, we have to find initial node with the lowest right-value (for the following axis) or the greatest left-value (for the preceding axis). Then we use this node as an input for simple version of axis resolution algorithm described in Section 3. Finding the initial node is trivial in case of the preceding axis, since the node with greatest left-value is always at the end of the set. In case of the following axis, situation is slightly more complicated. Node with the smallest right-value is either first node in the initial set or its descendant:

```
$i \leftarrow 1$
**while** $i < |S|$ *and* $r(S[i]) > r(S[i+1])$ **do**
  $i \leftarrow i + 1$
**end while**
return following nodes of $S[i]$
```

Almost every other axis may be accelerated as well using similar approach. We omit the description of the algorithms for the sake of scope. Complete overview can be found in [1].

4.3 Predicate Caches

Previous optimizations reduce necessary amount of work in many situations. However, the recursive algorithm still suffers from exponential time complexity. We shall avoid this problem using caches for predicate expressions. Problematic expressions (which are likely to be evaluated multiple times with the same context) will be *covered* by caches. If a cache covers an expression, all results of this expression are stored in cache (when first computed). When the expression is evaluated, the result is first looked up in the cache, thus each expression is executed for each context at most once.

The cache will reflect context class of the covered expression (e.g. expression from context-node class must be covered by context-node cache). The class of the cache defines which part of the context is used for indexing. Therefore, context-free cache stores single value, context-node cache is indexed by nodes and context-full cache utilizes context node, position and size. Full-context cache may consume up to $\mathcal{O}(N^4)$ memory (where N is size of the document); however, we may apply limits for cache size and make it forget records. Forgetful cache will not save us from exponential time complexity, but it still improves performance significantly.

Following rules are applied for caches:

- Literal values must not be covered by a cache directly.
- When an expression is covered by a cache, all its subexpressions from the same or stronger class are also considered to be covered. This rule is transitive and does not apply to predicates (a predicate cannot be covered transitively). All expressions from weaker classes must be covered by another cache of their own.

- If location step has a full-context predicate, all predicates of that step must be covered by caches. Otherwise, no predicate of that step needs to be covered, but these rules are applied recursively on nested predicates. Furthermore, when a predicate must be covered by cache, a cache with Boolean values is always used, considering the predicates are implicitly wrapped by *boolean*() function.

The cache is implemented by concurrent hash-map from TBB library [13]. This hash-map works as hash table with concurrent access with fine grade locking. Locking is implemented by atomic operations and each position in the table can be locked individually. Therefore, collisions on locks will be rare. Unfortunately, the results indicate that locking on caches has significant impact on efficiency.

5 Parallelization

We have described data structures and algorithms which should allow us to exploit parallelism using standard templates. These templates (parallel-for, parallel-reduce, parallel-scan, etc.) and data structures (concurrent vector, concurrent hash-map, etc.) are described in literature [12]. We omit their description for the sake of scope.

The implementation presents many opportunities to apply these templates and concurrent data structures. We describe only the most interesting ones. More detailed description is provided in related work [1].

5.1 Predicate Filtering

We expect to have list of predicates $\mathcal{P}_i$ $(i = 1 \dots \pi)$ and a node-set S being filtered. The filtering is processed as follows.

- First, all context-free predicates are resolved. Predicates resolved as *true* are immediately removed. When a single predicate is resolved as *false*, the whole filtering operation yields empty node-set since every node is exterminated by such predicate.
- Remaining predicates are grouped into *segments*. Full-context predicates must be positioned as first predicate in segment and there can be only one full-context predicate in each segment. Other predicates (from the context-node class) are grouped together to form as large segments as possible. For example, predicates[2] $nnfnfnn$ are grouped as $(nn)(fn)(fnn)$.
- Node-set S is filtered by first segment producing S', then S' is filtered by second segment and so on. Last segment produces the result of whole filtering operation.

[2] Where n denotes context-node predicate and f denotes full-context predicate.

The node-set is filtered by segment in two phases. In the first phase, all nodes from initial set are filtered by all predicates in the segment (the predicate order is respected). When a predicate is resolved as *false*, corresponding node in initial set is replaced by `null` value. In the second phase, non-`null` values are copied into filtered set.

The first phase utilizes parallel-for pattern. All nodes from the set are processed concurrently. Each task affects only its nodes, so no explicit synchronization is required. The second phase copies only valid nodes and must respect the ordering of the nodes. We use parallel-scan, where first pass (the pre-scan) only calculates new offsets for nodes (after `null`s are removed) and the second pass (final scan) copies the nodes.

5.2 Node-sets Merging

Some situations of XPath processing require merging node-sets (namely location steps with full-context predicates and the union operator). In both situations, a concurrent hash-map is used. All nodes are inserted into the hash-map (which ensures uniqueness of each node), then retrieved into an array and sorted. All three steps are performed in parallel fashion. Nodes that are inserted into hash-map are processed by parallel-for. The parallel-scan copies nodes into an array where first pass computes offsets and second pass copies the nodes. Finally, the array is sorted by parallel sort [12].

In case of union operator, we employ another parallel-for which processes operands of the union, so we have top level loop which operates over operands (and resulting node-sets) and nested loops which insert nodes into hash-map.

5.3 Processing Axes

There are hardly any opportunities for parallelism when an axis is processed for a single node. The only thing which can be done concurrently is copying sequence of nodes (using parallel-for). However, even when a location step is processed without optimizations, we may still process each node from initial set (i.e. resolve the axis and filter it by predicates) concurrently.

Vectorized processing of axes presents more opportunities for parallelism. In many cases (e.g. descendant axis, child axis, ...), node-sets produced by initial nodes are disjoint. Therefore, we need not use concurrent hash-map (which requires locking) to merge them. Instead, we use parallel-scan to assemble merged sets in correct order. The parallel scan is used as usual – first pass computes offsets for nodes and the second pass copies the nodes.

6 Practical Tests

Practical experiments were designed for two things. First, we would like to determine scalability of the solution and to measure speedup on multiple cores. Second, we compare our solution with existing libraries for XPath processing.

The comparison is relevant only for our implementation restricted to single core as both libxml and Xalan are single-threaded.

6.1 Methodology, Testing Data, Hardware Specifications

Performed tests will focus solely on execution speed. We will measure time required to evaluate a query using system real-time clock. Other operations such as loading XML data or processing results will not concern us. Real-time clock will better reflect the practical characteristics of the implementation and cover both application and kernel time (thus include context switching, thread synchronization, etc.).

Each query is executed 10×. *Raw average* is computed as arithmetic average of all 10 times. Then, each measured time which is greater than raw average multiplied by 1.25 (i.e. with 25% tolerance) is excluded. Final time is computed as arithmetic average of remaining values.

It is still possible that all ten measured values are distorted due to some long lasting activity running on the system at the same time as the tests. Therefore, each test-set was repeated three times in different daytime. These three results were closely compared and if one of the values was obviously tainted, the test was repeated. A result is considered tainted if it deviates from the other two by more than 25%.

Document used for testing was generated by xmlgen, a XML document generator developed under XMark [8] project. This document simulates an auction website (a real e-commerce application) and contains over 3 million elements.

Queries evaluated on the document are taken from XPathMark performance tests [9]. These queries are especially designed to determine speed of tested XPath implementation. Queries that were not compatible with our subset of XPath were omitted. Unfortunately, some of the results are not presented here due to lack of space. Complete data and results may be found in [1].

All test were performed on Dell M905 server with four six-core AMD Opterons 8431 (i.e. 24 cores) clocked at 2.4 GHz. Server was equipped with 96 GB of RAM organized in 4-node NUMA. A RedHat Enterprise Linux (version 5.4) was used as operating system.

6.2 Results

Table 2 summarizes times (in ms) measured for our implementation, libxml and Xalan libraries. Columns $\mathcal{C}_t$ show times required by our implementation on t threads. Symbol ∞ represents times that were completely out of scale[3].

The results suggest that our implementation is highly scalable. The best speedup was observed at query B_6 which runs 18.8× faster on 24 cores than on single core. Unfortunately, some queries (A_2, A_3, C_2, D_2, and E_4) have shown poor speedup or even slow-down. This is mostly caused by the structure of these queries. When intermediate results between location steps are too small,

[3] Times greater than 2 millions ms.

	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_4$	$\mathcal{C}_8$	$\mathcal{C}_{16}$	$\mathcal{C}_{24}$	libxml	Xalan
A_2	2.531	3.317	2.603	2.381	2.488	2.157	284.1	1054
A_3	2.816	3.520	2.562	2.43	2.366	2.127	312.5	305.8
B_5	629.9	337.1	208.9	135.3	85.38	80.94	∞	∞
B_6	5542	2809	1470	755.6	400	295.2	∞	∞
B_9	17830	9257	4769	2687	1654	1422	∞	∞
B_{10}	8955	4663	2396	1354	835.8	707.7	∞	∞
C_1	65.33	38.15	21.1	17.09	19.4	19.09	516.8	57.04
C_2	176.4	131.7	137.2	162.7	159.3	135.1	118.6	88.45
D_2	0.925	2.216	1.095	1.531	2.377	1.754	3922	45910
E_4	360.1	306.8	322.4	372.6	384.2	338.9	2891	369.9
E_5	6325	3304	1724	907.5	490.1	369.1	∞	∞

Table 2. Times in ms for XPathMark [9] tests

the query cannot benefit from data parallelism very much. Almost all of these queries (except for C_2 and E_4) are resolved very quickly on single core, therefore we need not parallelize them at all. Queries A_2, A_3, and D_2 are even slower on two cores than on single core. This is caused by locking overhead of caches. Finally, we have observed some fluctuations between times on 8, 16, and 24 cores which are most likely caused by effects of memory access on NUMA systems and thread distribution among real processors.

In comparison with known XPath processors [10][11], our implementation outperforms them in almost every test. The most significant difference is in queries B_5, B_6, B_9, B_{10}, and E_5 which all contain following or preceding axis. However, queries C_1 and C_2 are resolved slightly slower in our implementation, which is most likely caused by better optimizations of comparisons in libxml and Xalan.

7 Conclusions

This paper presents algorithms and indexing data structures for XPath processing which can be easily adapted by standard parallelization templates. Our experimental results demonstrate that proposed algorithms scale very well for long lasting queries. These algorithms are beneficial also for sequential processing as they outperform libxml and Xalan libraries in almost every test even on single core.

Our implementation uses some data structures which require locking. These structures are most likely responsible for poor speedup of some tested queries. Furthermore, our implementation does not analyze executed query nor data to determine whether particular part of the query should be parallelized or not. We use simple greedy method to partition workload and hope for the best.

We will focus on removing locking completely and try to design metrics for query evaluation cost in our future work.

References

1. M. Kruliš, J. Yaghob. Algorithms for Parallel Searching in XML Datasets, 2009. `http://www.ksi.mff.cuni.cz/~krulis/?page=study/master_thesis`
2. G. Gottlob, C. Koch, R. Pichler. Efficient algorithms for processing XPath queries. ACM Trans. Database Syst., 30(2):444491. ACM, New York, NY, USA, 2005.
3. Lu, W. and Gannon, D. Parallel XML Processing by Work Stealing SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches
4. Bordawekar, R. and Lim, L. and Shmueli, O. Parallelization of XPath queries using multi-core processors: challenges and experiences, EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology
5. Bordawekar, R. and Lim, L. and Kementsietsidis, A. and Kok, B.W.L. To Parallelize or Not to Parallelize: XPath Queries on Multi-core Systems
6. Clark, S. DeRose, et al. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16:1999, 1999.
7. XML Document Object Model. `http://www.w3.org/DOM/`
8. XMark - benchmark for various XML technologies. `http://www.xmlbenchmark.org/`
9. XPathMark - benchmark for XPath 1.0. `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/`
10. Libxml2 - The XML Library for GNOME. `http://xmlsoft.org/`
11. Xalan for C++ - XSLT and XPath library developed by Appache. `http://xml.apache.org/xalan-c/`
12. J. Reinders. Intel threading building blocks. OReilly & Associates, Inc., Sebastopol, CA, USA, 2007.
13. Intel Threading Building Blocks – an open source library for parallel programming. `http://www.threadingbuildingblocks.org/`.
14. Pan, Y. and Lu, W. and Zhang, Y. and Chiu, K. A static load-balancing scheme for parallel xml parsing on multicore cpus. IEEE International Symposium on Cluster Computing and the Grid, Rio de Janeiro, 2007
15. Wu, Yu, Qi Zhang, Zhiqiang Yu and Jianhui Li. "A Hybrid Parallel Processing for XML Parsing and Schema Validation." Presented at Balisage: The Markup Conference 2008, Montral, Canada, August 12 - 15, 2008. In Proceedings of Balisage: The Markup Conference 2008. Balisage Series on Markup Technologies, vol. 1 (2008). doi:10.4242/BalisageVol1.Wu01.
16. T. Grust. Accelerating XPath location steps. pages 109120. ACM, New York, NY, USA, 2002.
17. M. Kratky, J. Pokorny, and V. Snasel. Implementation of XPath axes in the multi-dimensional approach to indexing XML data. Current Trends in Database Technology-Edbt 2004 Workshops: EDBT 2004 Workshops, PhD, DataX, PIM, P2P&DB, and Clustweb, Heraklion, Crete, Greece, March 14-18, 2004: Revised Selected Papers, page 219. Springer, 2004.
18. I. Mlýnková, K. Toman, and J. Pokorný. Statistical Analysis of Real XML Data Collections.
In COMAD06: Proc. of the 13th Int. Conf. on Management of Data, pages 2031, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.